Technical Note

MAP 文件浅析

MDK 使用教程 - 正点原子团队编写

技术文档

修订历史

版本	日期	原因
V1.0	2019/6/30	第一次发布
V1.1	2020/5/16	改为最新版本例程格式进行说明

目录

1. MDK 编译生成文件简介 3
2. map 文件分析 7
2.1 map 文件的 MDK 设置 7
2.2 map 文件的基础概念 8
2.3 map 文件的组成部分说明 8
2.1.1 程序段交叉引用关系（Section Cross References） 9
2.1.2 删除映像未使用的程序段（Removing Unused input sections from the image） 9
2.1.3 映像符号表（Image Symbol Table） 10
2.1.3.1 本地符号（Local Symbols） 10
2.1.3.2 全局符号（Global Symbols） 11
2.1.4 映像内存分布图（Memory Map of the image） 11
2.1.5 映像组件大小（Image component sizes） 13
3. 其他 16

1. MDK 编译生成文件简介

MDK 编译工程，会生成一些中间文件（如.o、.axf、.map 等），最终生成 hex 文件，以便下载到 MCU 上面执行，以 MiniPRO STM32H750 开发板的跑马灯实验（实验 1）为例（其他开发板例程类似），编译过程产生的所有文件，都存放在 Output 文件夹下（对应老版本例程的 OBJ 文件夹），如图 1.1 所示：

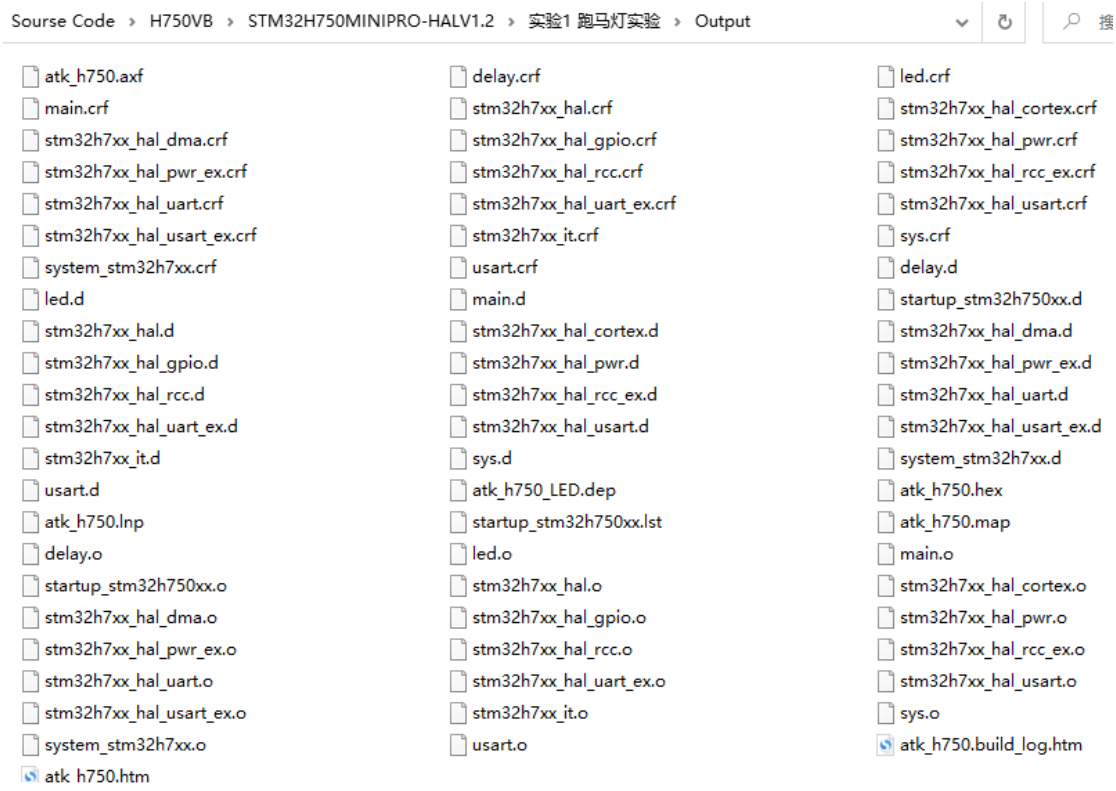


图 1.1 MDK 编译过程生成的文件

可以看到，这里总共生成了 67 个文件，分为 11 个类型，分别是：.axf、.crf、.d、.dep、.hex、.lnp、.lst、.o、.htm、bulild_log.htm 和.map。67 个文件看着还不是很多，但是随着工程的增大，这些文件也会越来越多，大项目编译一次，可以生成几百甚至上千个这种文件，不过文件类型基本就是上面这些。

对于 MDK 工程来说，基本上任何工程在编译过程中都会有这 11 类文件，常见的 MDK 编译过程生产文件类型如表 1.1 所示：

文件类型	说明
.o	可重定向[1]对象文件，每个源文件（.c/.s 等）编译都会生成一个.o 文件
.axf	由 ARMCC 编译生产的可执行对象文件，不可重定向[2]（绝对地址） 多个.o 文件链接生成.axf 文件，我们在仿真的时候，需要用到该文件
.hex	Intel Hex 格式文件，可用于下载到 MCU，.hex 文件由.axf 文件转换而来
.crf	交叉引用文件，包含浏览信息（定义、标识符、引用）
.d	由 ARMCC/GCC 编译生产的依赖文件（.o 文件所对应的依赖文件） 每个.o 文件，都有一个对应的.d 文件
.dep	整个工程的依赖文件
.lnp	MDK 生成的链接输入文件，用于命令输入
.lst	C 语言或汇编编译器生成的列表文件

.htm	链接生成的列表文件
.build_log.htm	最近一次编译工程时的日志记录文件
.map	连接器生成的列表文件/MAP 文件，**该文件对我们非常有用！**

表 1.1 常见的中间文件类型说明

注 1，可重定向是指该文件包含数据/代码，但是并没有指定地址，它的地址可由后续链接的时候进行指定。

注 2，不可重定向是指该文件所包含的数据/代码都已经指定地址了，不能再改变。

表 1.1 中，加粗的 5 种文件类型，对我们学习来说比较关心，尤其是.map 文件，是本文档需要重点介绍的。这里我们先简单说一下这 5 种文件的作用。

.o 文件，它是由编译器编译.c/.s 文件时所产生的可重定向对象文件，其文件名同.c/.s 文件一模一样，只是后缀为.o。所以，我们看到.o 就应该想到与之对应的.c 或者.s 文件，我们在分散加载里面经常会用到.o 文件。

.axf 文件，它是由 armlink 链接器，将整个工程参与编译的.o 文件链接成一个可执行对象文件，它是不可重定向的。有了该文件，我们就可以用仿真器来下载到 MCU 进行仿真调试了。注意：各类仿真器，在进行下载调试的时候，都是使用的.axf 文件。

.hex 文件，它是由.axf 转换而来的一个可执行对象文件。.hex 文件和.bin 文件的区别是：.bin 文件不含地址信息，全部都是可执行代码；而.hex 文件则是包含地址信息的可自行代码。同样的.bin 文件也是由.axf 文件转换而来的。我们在使用 ISP 软件进行程序下载的时候，一般使用的是.hex 文件，由 ISP 软件解析.hex 文件包含的地址信息来实现程序下载。而我们在进行 BootLoader 升级的时候，一般使用.bin 文件，地址由 Bootloader 程序指定。

.htm 文件，它是编译器在编译代码的时候生成的一个列表文件，包含了整个工程的静态调用图，最大的用处就是可以查看栈深度（最小深度），方便设置栈大小。.htm 文件可以直接由浏览器打开（双击打开）。.htm 文件包含两部分内容：

1， 整个工程最大栈（Stack）深度及其调用关系

我们打开：新建工程例程→Output→atk_h750.htm 文件（双击，注意：必须整个工程编译一遍，才会生成 atk_h750.htm 文件，否则是找不到这个文件的！），可以看到如图 1.2 所示：

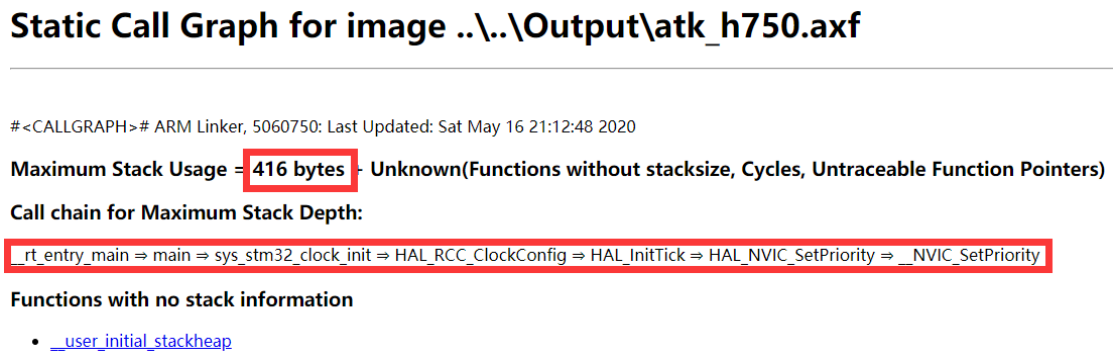


图 1.2 .htm 文件显示最大栈深及调用关系

可以看到，例程的最大栈深度是 416 字节，最大栈深时的调用关系为：__rt_entry_main ⇒ main ⇒ sys_stm32_clock_init ⇒ HAL_RCC_ClockConfig ⇒ HAL_InitTick ⇒ HAL_NVIC_SetPriority ⇒ __NVIC_SetPriority。

不过需要注意的是，这里的最大栈深度仅仅是最低要求（静态栈），因为它并没有统计无栈深的函数（用内存管理）、递归函数、以及无法追踪的函数（函数指针）等所包含的栈（Stack）。

不过它给我们指明了最低需求，我们在分配栈深度的时候，就可以参考这个值来做设置，一般不低于静态栈的 2 倍。我们例程默认设置的栈深(Stack_Size)为 0X800（通过.s 文件设置），

即 2048 字节，如图 1.2 所示：

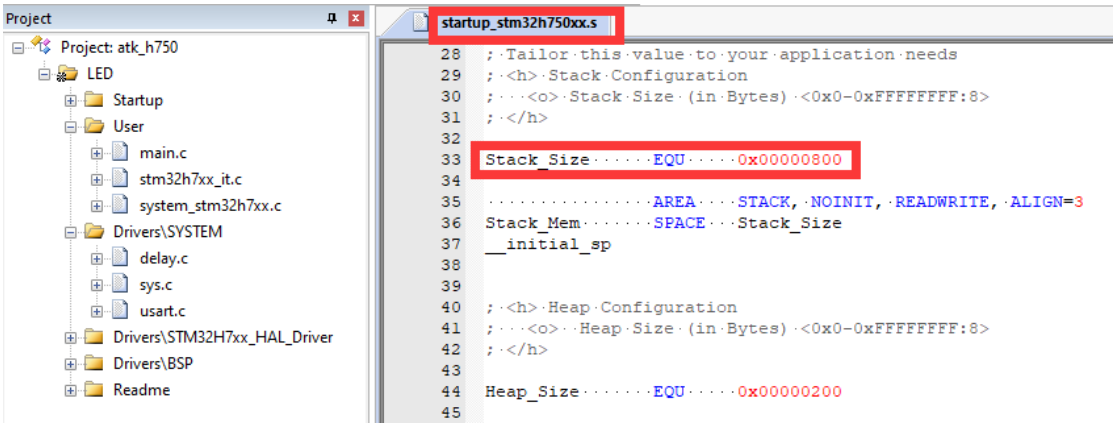


图 1.2 栈深度（大小）设置

2， 各个函数的栈深及其调用关系

.htm 文件还给出了每个函数所使用的栈深度以及其调用关系，如图 1.3 所示：

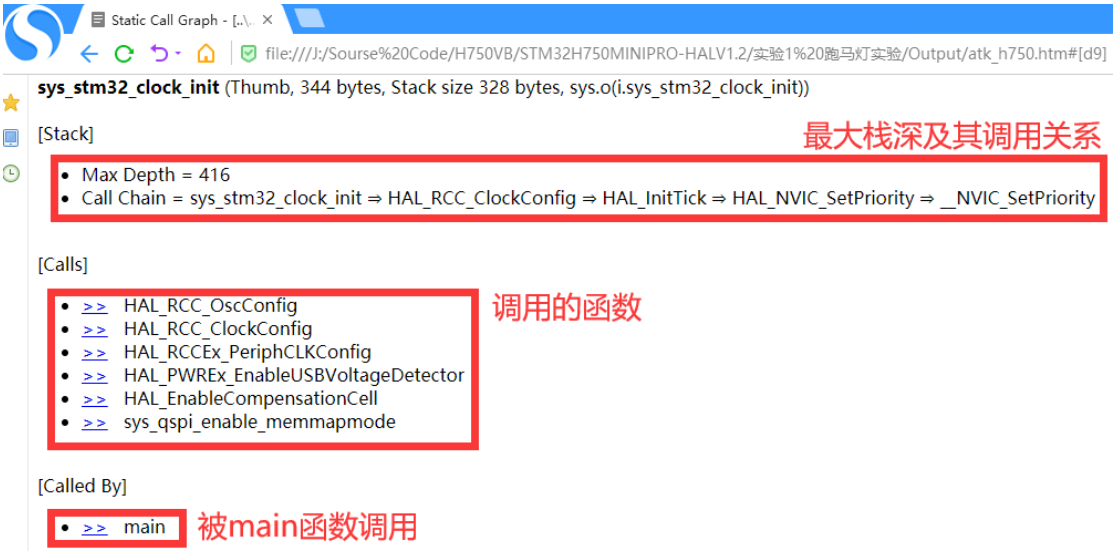


图 1.2 单个函数栈深度及其调用关系

上图给出了 sys_stm32_clock_init 函数的最大栈深度及其调用关系，并且列出了其所调用的函数及其所被调用的函数。

.map 文件，对我们编程非常有帮助，是本文档重点要给大家介绍的，因此另起一节，进行重点说明。

其他文件类型及说明，请大家参考：MDK→help→uVision Help→B. File Types，如图 1.3 所示：

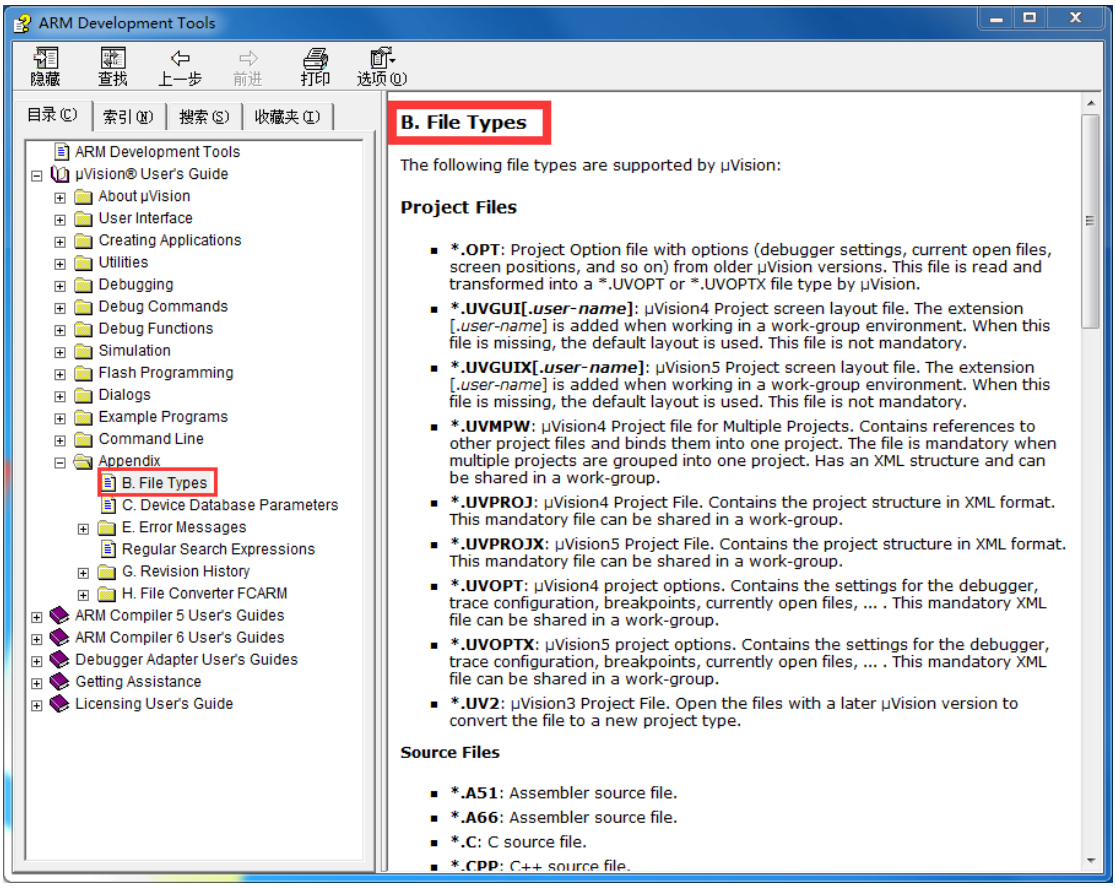


图 1.3 MDK uVision Help 查看文件类型说明

2. map 文件分析

.map 文件是编译器链接时生成的一个文件，它主要包含了交叉链接信息。通过.map 文件，我们可以知道整个工程的函数调用关系、FLASH 和 RAM 占用情况及其详细汇总信息，能具体到单个源文件（.c/.s）的占用情况，根据这些信息，我们可以对代码进行优化。

.map 文件可以分为以下 5 个组成部分：

1， 程序段交叉引用关系（Section Cross References）
2， 删除映像未使用的程序段（Removing Unused input sections from the image）
3， 映像符号表（Image Symbol Table）
4， 映像内存分布图（Memory Map of the image）
5， 映像组件大小（Image component sizes）

接下来，我们将分三个部分对 map 文件进行详细介绍：

1， map 文件的 MDK 设置
2， map 文件的基础概念
3， map 文件的组成部分说明

2.1 map 文件的 MDK 设置

要生成 map 文件，我们需要在 MDK 的魔术棒→Listing 选项卡里面，进行相关设置，如图 2.1.1 所示：

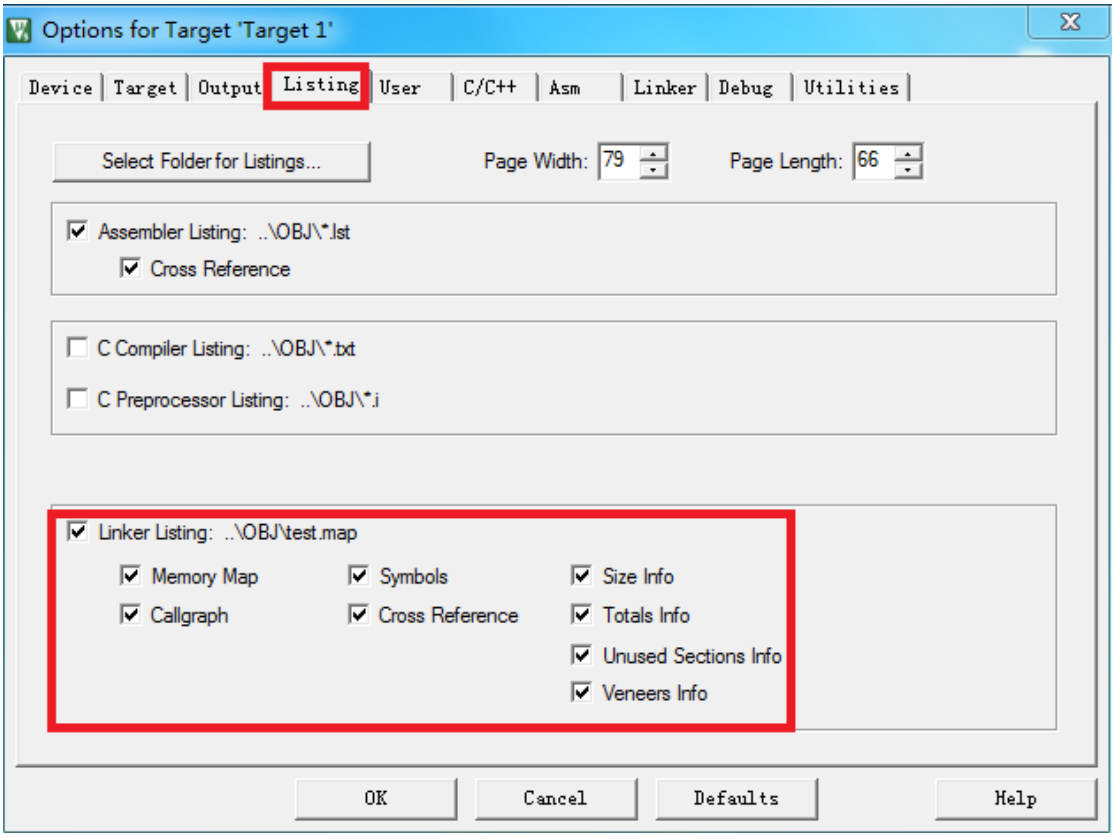


图 2.1.1 .map 文件生成设置

图 2.1.1 中红框框出的部分就是我们需要设置的，默认情况下，MDK 这部分设置就是全勾选的，如果我们想取消掉一些信息的输出，则取消相关勾选即可（一般不建议）。

如图 2.1.1 设置好 MDK 以后，我全编译当前工程，当编译完成后（无错误），就会生成.map 文件。在 MDK 里面打开.map 文件的方法如图 2.1.2 所示：

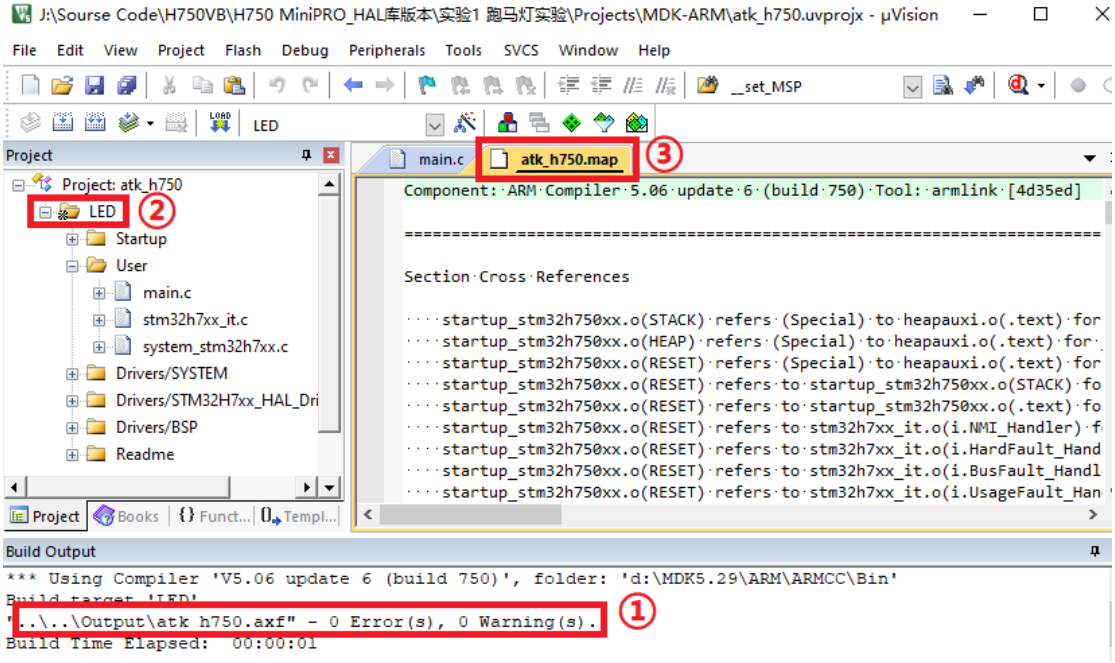


图 2.1.2 打开.map 文件

① ，先确保工程编译成功（无错误）。

② ，双击工程目标 LED（不同工程名字可能不一样），打开.map 文件。

③ ，map 文件打开成功。

2.2 map 文件的基础概念

为了更好的分析 map 文件，我们先对需要用到的一些基础概念进行一个简单介绍，相关概念如下：

- Section：描述映像文件的代码或数据块，我们简称程序段
- RO：Read Only 的缩写，包括只读数据（RO data）和代码（RO code）两部分内容，占用 FLASH 空间
- RW：Read Write 的缩写，包含可读写数据（RW data，有初值，且不为 0），占用 FLASH（存储初值）和 RAM（读写操作）
- ZI：Zero initialized 的缩写，包含初始化为 0 的数据（ZI data），占用 RAM 空间。
- .text：相当于 RO code
- .constdata：相当于 RO data
- .bss：相当于 ZI data
- .data：相当于 RW data

2.3 map 文件的组成部分说明

我们前面说了，map 文件分为 5 个部分组成，接下来分别介绍这个 5 个部分。这里还是以 MiniPRO STM32H750 开发板的跑马灯实验（实验 1）为例进行介绍，其他开发板请参考着学习。

2.1.1 程序段交叉引用关系（Section Cross References）

这部分内容描述了各个文件（.c/.s 等）之间函数（程序段）的调用关系，如图 2.1.1.1 所示：

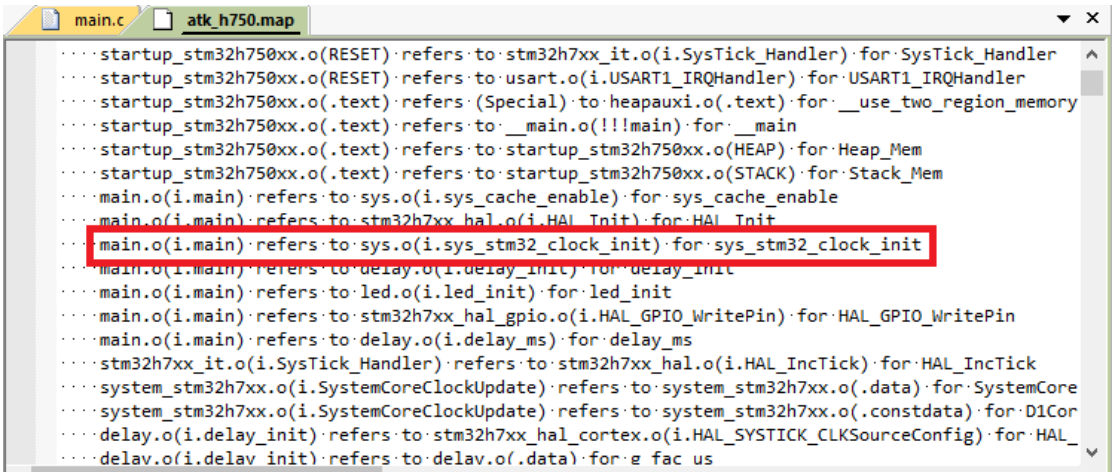


图 2.1.1.1 程序段交叉引用关系图

上图中，红框框出的部分：main.o(i.main) refers to sys.o(i.sys_stm32_clock_init) for sys_stm32_clock_init，表示：main.c 文件中的 main 函数，调用了 sys.c 中的 sys_stm32_clock_init 函数。其中：i.main 表示 main 函数的入口地址，同理 i.sys_stm32_clock_init 表示 sys_stm32_clock_init 的入口地址。

2.1.2 删除映像未使用的程序段（Removing Unused input sections from the image）

这部分内容描述了工程中由于未被调用而被删除的冗余程序段（函数/数据），如图 2.1.2.1 所示：

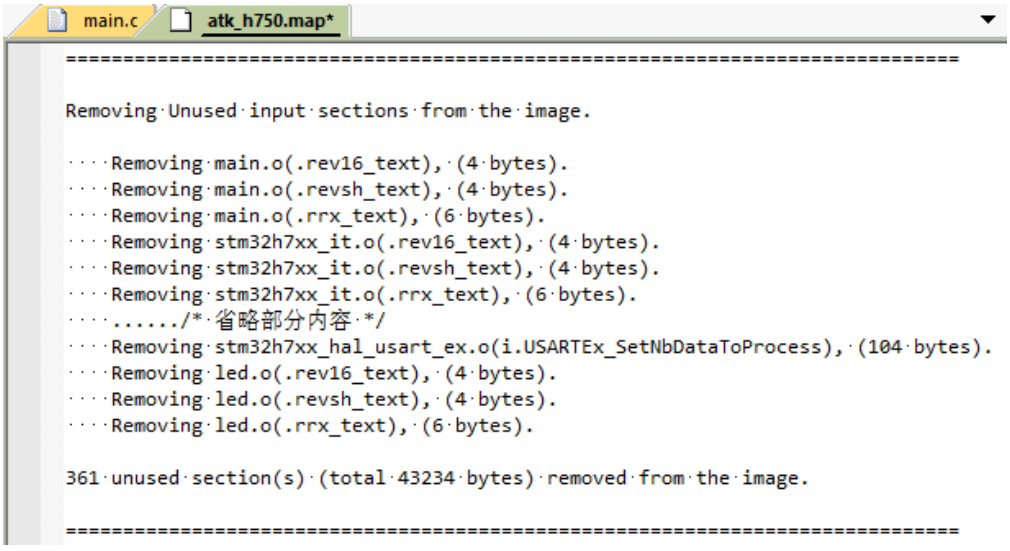


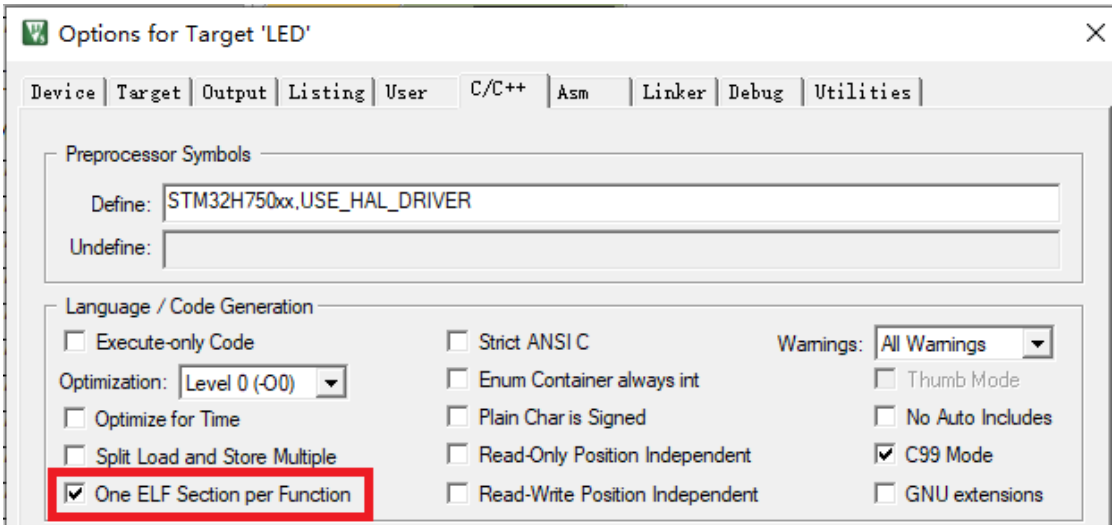
图 2.1.2.1 删除未用到的程序段

上图中，列出了所有被移除的程序段，比如 stm32h7xx_hal_usart_ex.c 文件里面的

USARTEx_SetNbDataToProcess 函数就被移除了，因为该例程没用到这个函数。

另外，在最后还有一个统计信息： 361 unused section(s) (total 43234 bytes) removed from the image. 表示总共移除了 361 个程序段（函数/数据），大小为 43234 字节。即给我们的 MCU 节省了 43234 字节的程序空间。

为了更好的节省空间，我们一般在 MDK→魔术棒→C/C++选项卡里面勾选：One ELF Section per Function，如图 2.1.2.2 所示：



2.1.2.2 MDK 勾选 One ELF Section per Function

2.1.3 映像符号表（Image Symbol Table）

映像符号表（Image Symbol Table）描述了被引用的各个符号（程序段/数据）在存储器中的存储地址、类型、大小等信息。映像符号表分为两类：本地符号（Local Symbols）和全局符号（Global Symbols）。

2.1.3.1 本地符号（Local Symbols）

本地符号（Local Symbols）记录了用 static 声明的全局变量地址和大小，c 文件中函数的地址和用 static 声明的函数代码大小，汇编文件中的标号地址（作用域：限本文件），本地符号如图 2.1.3.1.1 所示：

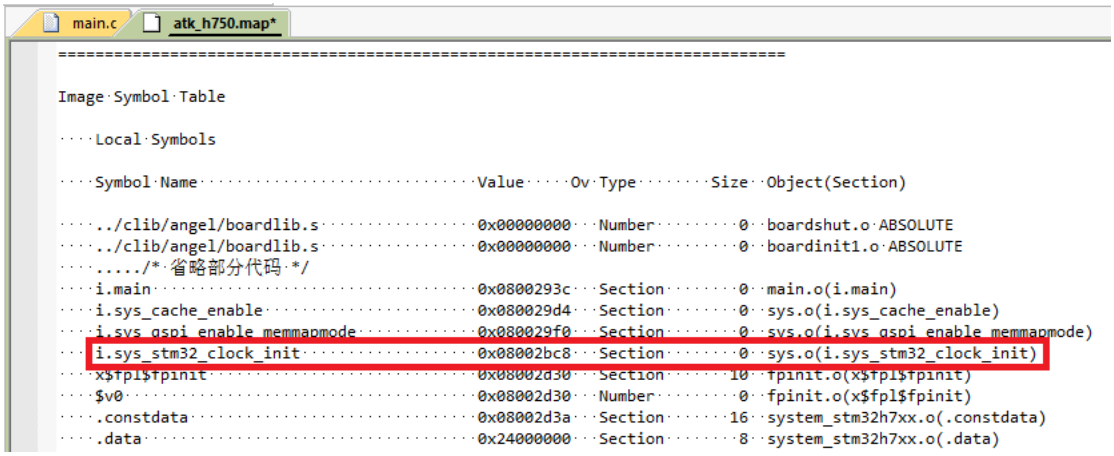


图 2.1.3.1.1 本地符号（Local Symbols）

图中红框框处部分，表示 sys.c 文件中的 sys_stm32_clock_init 函数的入口地址为：

0x08002bc8，类型为：Section（程序段），大小为 0。因为：i. sys_stm32_clock_init 仅仅表示 sys_stm32_clock_init 函数入口地址，并不是指令，所以没有大小。在全局符号段，会列出 sys_stm32_clock_init 函数的大小。

2.1.3.2 全局符号（Global Symbols）

全局符号（Global Symbols）记录了全局变量的地址和大小，C 文件中函数的地址及其代码大小，汇编文件中的标号地址（作用域：全工程），全局符号如图 2.1.3.2.1 所示：

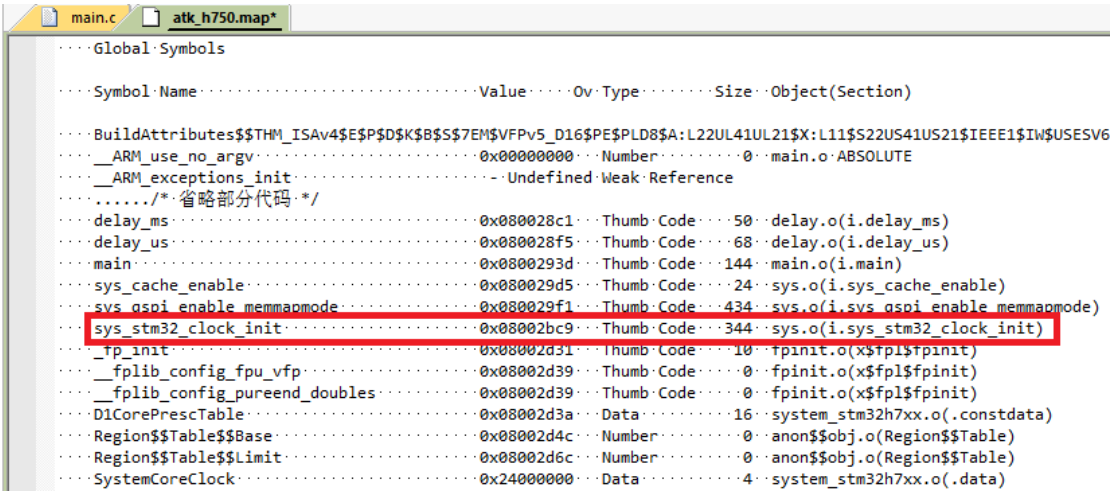


图 2.1.3.2.1 本地符号（Global Symbols）

图中红框框处部分，表示 sys.c 文件中的 sys_stm32_clock_init 函数的入口地址为：0x08002bc9，类型为：Thumb Code（程序段），大小为 344 字节。

注意，此处的地址用的 0x08002bc9，和 2.1.3.1 节的 0x08002bc8 地址不符，这是因为 ARM 规定 Thumb 指令集的所有指令，其最低位必须为 1，0x08002bc9 = 0x08002bc8 + 1，所以才会有 2 个不同的地址，且总是差 1，实际上就是同一个函数。

2.1.4 映像内存分布图（Memory Map of the image）

映像文件分为加载域（Load Region）和运行域（Execution Region），一个加载域必须有至少一个运行域（可以有多个运行域），而一个程序又可以有多个加载域。加载域为映像程序的实际存储区域，而运行域则是 MCU 上电后的运行状态。加载域和运行域的简化关系（这里仅表示一个加载域的情况）图如图 2.1.4.1 所示：

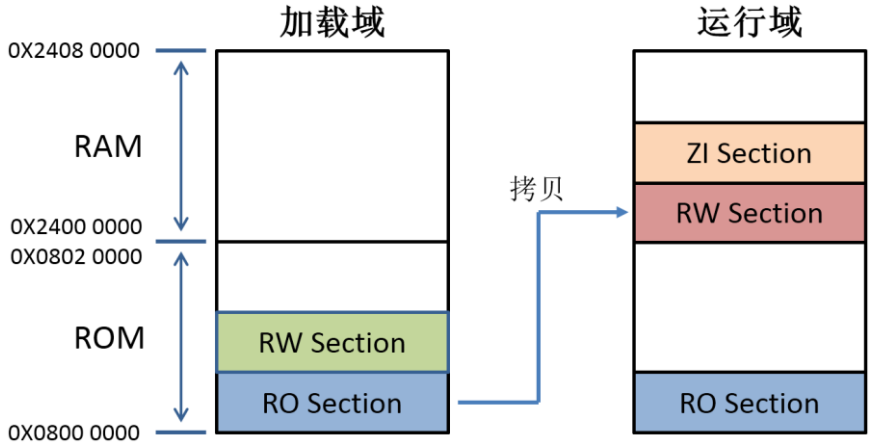


图 2.1.4.1 加载域运行域关系

由图可知，RW 区也是存放在 ROM（FLASH）里面的，在执行 main 函数之前，RW（有初值且不为 0 的变量）数据会被拷贝到 RAM 区，同时还会在 RAM 里面创建 ZI 区（初始化为 0 的变量）。

了解了加载域和运行域的作用及关系，我们再来看映像内存分布图（H750 例程），如图 2.1.4.2 所示：

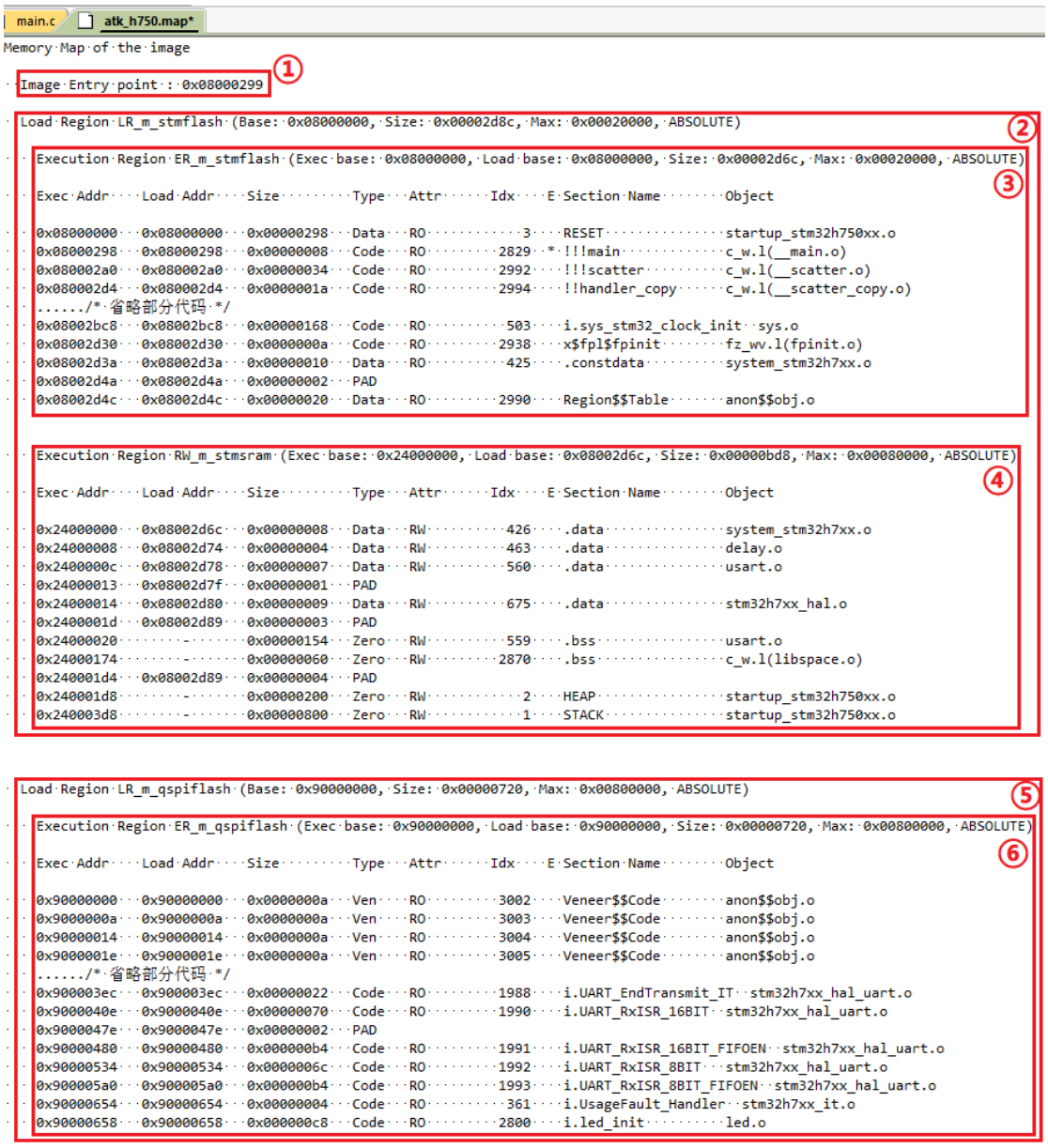


图 2.1.4.2 映像内存分布图

① 处，表示映像的入口地址，也就是整个程序运行的起始地址，为：0X0800 0299。实际地址为：0X0800 0298（Thumb 指令最低位是 1）。

② 处，表示 LR_m_stmflash 加载域，其起始地址为：0X0800 0000；占用大小为：0X0000 2D8C；最大地址范围为：0X0002 0000。其内部包含两个运行域：ER_m_stmflash 和 RW_m_stmsram。

③ 处，表示 ER_m_stmflash 运行域，其起始地址为：0X0800 0000；占用大小为：0X0000 2D6C；最大地址范围为：0X0002 0000；即内部 FLASH 运行域，所有需要放内部 FLASH 的代码，都应该放到这个运行域里面。对于 STM32F1/F4/F767 等开发板，

我们例程所有的代码，都是放在这个运行域的（名字可能不一样）。

④ 处，表示 RW_m_stmsram 运行域，其起始地址为：0X2400 0000；占用大小为：0X0000 2D6C；最大地址范围为：0X0008 0000；即内部 SRAM 运行域，所有 RAM（包括 RW 和 ZI）都是放在这个运行域里面。

⑤ 处，表示 LR_m_qspiflash 加载域，其起始地址为：0X9000 0000；占用大小为：0X0000 0720；最大地址范围为：0X0080 0000。其内部包含一个运行域：ER_m_qspiflash。

⑥ 处，表示 ER_m_qspiflash 运行域，其起始地址为：0X9000 0000；占用大小为：0X0000 0720；最大地址范围为：0X0080 0000；即外部 QSPI FLASH 运行域，所有需要放外部 QSPI FLASH 的代码，都应该放到这个运行域里面。

图 2.1.4.2 中，列出了所有加载域及其运行域的具体内存分布，我们可以很方便的查看任何一个函数所在的运行域、入口地址、占用空间等信息。如 sys_stm32_clock_init 函数：该函数在 ER_m_stmflash 运行域；入口地址为：0X0800 2BC8；大小为：0X168 字节；是 sys.c 里面的函数。了解这些信息，对我们分析及优化程序非常有用。

2.1.5 映像组件大小（Image component sizes）

映像组件大小（Image component sizes）给出了整个映像所有代码（.o）占用空间的汇总信息，对我们比较有用，如图 2.1.5.1 所示：

main.c | atk_h750.map*

```
Image component sizes

  Code (inc. data)   RO Data   RW Data   ZI Data      Debug   Object Name

     142          8          0          4          0       2247   delay.o
     200         12          0          0          0       1067   led.o
     152          8          0          0          0    1218522   main.o
      96         30        664          0       2560       1012   startup_stm32h750xx.o
      ...../* 省略部分代码 */
    1140         74          0          0          0      39482   sys.o
       0          0         16          8          0        961   system_stm32h7xx.o
     220         30          0          7        340       3941   usart.o

   12456        480        714         32       2900    1362905   Object Totals
      90          0         32          0          0          0   (incl. Generated)
       4          0          2          4          0          0   (incl. Padding)

  Code (inc. data)   RO Data   RW Data   ZI Data      Debug   Library Member Name

       8          0          0          0          0         68   __main.o
       0          0          0          0          0          0   __rtentry.o
      12          0          0          0          0          0   __rtentry2.o
      ...../* 省略部分代码 */
       2          0          0          0          0         68   use_no_semi_2.o
      10          0          0          0          0        116   fpinit.o

     282         12          0          0        100        700   Library Totals
       6          0          0          0          4          0   (incl. Padding)

  Code (inc. data)   RO Data   RW Data   ZI Data      Debug   Library Name

     266         12          0          0         96        584   c_w.l
      10          0          0          0          0        116   fz_wv.l

     282         12          0          0        100        700   Library Totals

==============================================================================

  Code (inc. data)   RO Data   RW Data   ZI Data      Debug

   12738        492        714         32       3000    1358249   Grand Totals
   12738        492        714         32       3000    1358249   ELF Image Totals
   12738        492        714         32          0          0   ROM Totals

==============================================================================

Total RO  Size (Code + RO Data)                13452 (  13.14kB)
Total RW  Size (RW Data + ZI Data)              3032 (   2.96kB)
Total ROM Size (Code + RO Data + RW Data)      13484 (  13.17kB)

==============================================================================
```

图 2.1.5.1 映像组件大小

上图中，框出的三处信息对我们比较有用，接下来分别介绍：

① 处，表示.c/.s 文件生成对象所占空间大小（单位：字节，下同），即.c/.s 文件编译后所占代码空间的大小。每个项所代表的意义如下：

Code（inc.data）：表示包含内联数据（inc.data）后的代码大小。如 delay.o（即 delay.c）所占的 Code 大小为 142 字节，其中 8 字节是内联数据。

RO Data：表示只读数据所占的空间大小，一般是指 const 修饰的数据大小。

RW Data：表示有初值（且非 0）的可读写数据所占的空间大小，它同时占用 FLASH 和 RAM 空间。

ZI Data：表示初始化为 0 的可读写数据所占空间大小，它只占用 RAM 空间。

Debug：表示调试数据所占的空间大小，如调试输入节及符号和字符串。

Object Totals：表示以上部分链接到一起后，所占映像空间的大小。

(incl.Generated)：表示链接器生产的映像内容大小，它包含在 Object Totals 里面了，这里仅仅是单独列出，我们一般不需要关心。

（incl.Padding)：表示链接器根据需要插入填充以保证字节对齐的数据所占空间的大小，它也包含在 Object Totals 里面了，这里单独列出，一般无需关心。

② 处，表示被提取的库成员（.lib）添加到映像中的部分所占空间大小。各项意义同①中的说明。我们一般只用看 Library Totals 来分析库所占空间的大小即可。

③ 处，表示本工程全部程序汇总后的占用情况。其中：

Grand Totals：表示整个映像所占空间大小。

ELF Image Totals：表示 ELF 可执行链接格式映像文件的大小，一般和 Grand Totals 一样大小。

ROM Totals：表示整个映像所需要的 ROM 空间大小，不含 ZI 和 Debug 数据。

Total RO Size：表示 Code 和 RO 数据所占空间大小，本例程为：13452 字节。

Total RW Size：表示 RW 和 ZI 数据所占空间大小，即本映像所需 SRAM 空间的大小，本例程为：3032 字节。

Total ROM Size：表示 Code、RO 和 RW 数据所占空间大小，即本映像所需 FLASH 空间的大小，本例程为：13484 字节。

图 2.1.5.1 中，我们未框出的：Library Name 部分，实际和②处是一个意思，只是 Library Name 说明了②处的那些.o 文件来自什么库，这里实际上就是：fpinit.o 来自 fz_wv.l 库，其他部分来自 c_w.l 库。fz_wv.l 和 c_w.l 是库名字。

MAP 文件的分析就给大家介绍到这里。

3. 其他

1、 购买地址：

官方店铺 1：https://openedv.taobao.com

官方店铺 2：https://zhengdianyuanzi.tmall.com

2、 资料下载

模块资料下载地址：http://www.openedv.com/docs/index.html

3、 技术支持

公司网址：www.alientek.com

技术论坛：www.openedv.com

在线教学：www.yuanzige.com

传真：020-36773971

电话：020-38271790

正点原子